UiO : Department of Informatics
University of Oslo

Automated web-cache configuration using machine learning

Mads Larsen

Master's Thesis Spring 2014

Automated web-cache configuration using machine learning

Mads Larsen

20th May 2014

Abstract

As the Internet today grows together with user and content the Internet traffic also increases. The World Wide Web (WWW) is one of the most popular applications used on the Internet today, and with this it experiences network congestion and overloading of the original web server. To counter this growing issue web caches are used to reduce network traffic and web server requests between clients and web servers. Implementation, adoption and usage of web caches can be time consuming for sysadmins but a necessity. In this paper we will explore the possibilities of automating the web caches using machine learning techniques to reduce sysadmin workload and errors making web caches more adoptable.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude and appreciation to my supervisor, Kyrre Begnum. He has been the heart of my thesis, a great mentor, teacher and showed great dedication on helping me complete this thesis. I feel it couldn't been achieved without him.

Secondly i would like to thank Anis Yazidi for feedback and knowledge around machine learning. His help in this area has given me a better understanding and insight into what machine learning is all about.

I would also like to thank my fellow master students for helping me in many different ways to complete my goal. They gave me great advice and technical help and also helped me keep my spirit up trough out this master thesis.

Mads Larsen

Chapter 1

Introduction

1.1 Motivation

Since the launch of the World wide web (WWW) websites have grown exponentially, becoming an integral part of personal and business life. With the growth of web sites, huge amount of time and resources have been focused on web servers. In the early days after the launch of the world wide web, web servers were quite expensive and still are compared to what they have to serve today regarding Internet content. Since websites today contain a lot of data, web servers can become overloaded trying to serve the increasing amount of users on the Internet today.

Since the Internet has grown in popularity because users can do so much via the web today it has become a integral part of their life. People are depending on it more and more, meaning it has to be a 24/7 available service which also includes your own website. With the increase of mobile devices connected to the Internet and IPv6 implementation allowing indefinitely amount of devices connected to the Internet, the growth and increase of HTTP traffic will only continue.

Most web servers today have their limitations, the challenges with these limitations is that when reached, your website is already in trouble since reaching a web servers saturation point often means failure to serve the given website to users at all. The reason for this is that a web server needs to build each web page request before it can send it to the users. This can be quite a lot of work for a web server since web sites today contain more and more data with increased functionality. A web server may need to get content from other server like databases, SAN's etc.

Reaching the saturation point of a web server is something that is expected not to happen today, users today expect their website to be available 24/7. If a web site is not available when a user tries to access it at that specific moment, this user will in most cases not add value to the organisation. This is a crucial part on how the web is perceived by users, it should just work all the time.

To help counter web servers reaching their saturation point, web caching is a common solution. A web cache is a reverse proxy that is put in front of a web server between the users and the web server. The web cache works by taking a copy of a processed web page from the web server, stores it and forwards it directly to the users. If a page is often requested, this page is now served via the web cache to the users offloading the web servers.

The benefits of using a web cache are potentially quite huge, according to [1], [2] it can help speed up web sites by 80 percent depending on the architecture. By offloading web servers by a huge margin means that less web servers are needed to serve the same amount of users. It also helps reduce latency since users don't have to wait for the web server to process their request since they can get a copy of it from the web cache. This also helps reduce bandwidth to the web servers on the internal networks, meaning you would need less network equipment to handle the network load. Network distance to users can also be reduced by using a web cache, since it could be geographically located closer to the users.

There are other ways to help a web server; cookies on client sides help reduce web server load by remembering stateful information from websites. This helps because the server doesn't need to look for the same information each time the user revisits the web site. However cookies are though not a good way to cache since they only exist on the user side on one machine, cookies also get deleted, removed etc. They can also be somewhat misused to track peoples Internet usage.

Furthermore, there are also many other ways cookies can be used by hackers and attackers like network eavesdropping and cookie theft. These are some of the drawbacks with cookies since a cookie doesn't really identify a person but a web browser, which can have multiple users.

Since Internet users today have become so reliant on websites and Internet applications it's crucial that websites work when a user wants to access it. This is more important today and can cause a lot of lost business since users come from all around the world from different time zones etc. The web sites need to be available 24/7, and be able to handle spikes of new users when ,say, releasing a new product. This is where a website really benefits from web caches.

However web caches can be difficult and time consuming for sysadmins to setup and manage 24/7. A web cache needs to be told what to cache, for how long and many other options that needs to be taken into account. The web cache also needs to be somewhat tweaked and managed as the world and systems expand and change, this consumes a lot of a sysadmins time. Solutions to this are of course using third party web cache company's to handle this, however this can be quite expensive. Hiring more sysadmins to manage your own web caches also cost more money. Having to manage these web caches by humans who do make mistakes. Training

new sysadmins to manage web caching is also time consuming and can be expensive.

If one could automate web caching somehow using machine learning techniques this could really help reduce web cache management, configuration errors, complexity, increase adoption of web caches and in turn reduce sysadmin workload, freeing up sysadmin to do more important things to take the company forward in many other areas.

The impact of automating the a web cache would make it more adoptable for company's hesitating about not having a enough time and money to implement it.

More web caching implementation will help company's save money by reducing web servers, power usage, sysadmins salary and improve web site QoS for customers. Customers having a good web site experience will keep them happier which in turn means more customers and more traffic which means more users and more money.

Since web caches does not automate the process of what to cache, this can be time consuming and make it difficult for sysadmins to implement and use a web cache. In this thesis exploring the possibility of automating this process to reduce sysadmin management would make web cache easier to use and reduce sysadmin workload on previous and future setups. It would also make web caches more adoptable for new companies wanting to implement a web cache.

The goal of this paper would be to see if one can automate a web cache by using machine learning techniques. How much time would one save setting it up, configuring and managing it?

1.2 Problem Statement

Q1 - How automated web-cache configuration using machine learning could be achived

Automation is a great way to save money, time and reduce errors. But automating is not always an easy thing to do, it requires careful planning, knowledge, technical understanding and a birds eye view of what one is trying to automate.

Automation is a something a system administrator should always try to achive, a golden rule for system administrators is "if you can automate it, do it". With this in mind one could imagine a web cache configuration beeing automated. Since the configuration part of a web cache is something that needs to be done each time with continious configuration, eliminating this part would greatly save time and reduce errors.

Web caches can be used in different ways but in this thesis a reverse proxy HTTP caches will be used. This means it only supports uncrypted HTTP traffic and will always be setup as a reverse proxy.

Configurations
Almost all software used in a Linux environment needs some configuration, either via setup or after setup to make it work in the specific environment with own parameters. Configuration can be tidious, time consuming and hard to do and often needs to be tested.

If the process of continously configurating a web cache can be eliminated one could save a lot of time, reduce errors and free up sysadmin to do other work. It will also give a sort of confidence that it will work rather then doing it manualy and including human error.

Machine learning Machine learning in short means a system can change its behavior while learning from data. In this thesis learning from data is key, web traffic will be used as the data from whom the system will base its decisions on. When learning from data it will use this information to alter values in a ranking system using different preset algorithms that will be made during this thesis.
This means that while web traffic data changes the ranking system will change its values accordlingly to adapt, chaning the systems behavior.

Chapter 2

Background and literature

The background chapter will cover different technologies and aspects needed to achive automated web-cache configuration using machine learning. Since different technoligies will be used in this paper some core understanding of each them is needed. The background chapter will also help give a bigger picture of what and why this paper is trying to achive. It will cover fields ranging from internet technology, system administrators and some ideology around web caching.

2.1 HTTP

HTTP stands for Hypertext Transfer Protocol, it was created by W3C and IETF and is a part of the Internet protocol suit. Its main responsibility is the World Wide Web. Its a essential part of the Internet protocol suit and for the Internet surfing to work. The HTTP protocol is used via a Web browser to talk to web servers and ask for documents (web sites) using an URL (Uniform resource locator) like this http://www.varnish-software.com/. When a client request a web page it must know the URL, either via a search engine (google).

Often combined with HTML to serve dynamic content is CGI (Common gateway interface). It is basically scripts that help serve dynamic content between a web server and applications.

The HTTP GET option is the most commonly used option in HTTP, its purpose is to simply retrive data and nothing more. This option will be a key part of this thesis since web caches and a web server mostly only use this option, simply becasue reverse proxy web caches and web servers only retrives request.

HTTP Head is another option used to only reatrive meta-information and not the content.

Tough more HTTP options exist, they are not relevant to this thesis.

2.1.1 URL

The uniform resource locator is part of a the URI (Uniform resource identifer) which also includes the URN (Uniform resource name). URN can be used to identify books like ISBN 0-4556-32323-4 for example but regarding the HTTP protocol, URL is used to to identify web addresses.

A URL is used in the following format http://example.com/examplepage.html where the **/examplepage.html** would be the URL. This part is what would show up in web server logs including other information about the client.

The URL part is important in this thesis since it will be used to identify mostly used URL's that needs to be cached using varnish.

2.2 Internet Trends

Internet today is a global high availability service that the world has become depended on, with this the number of users have greatly increased. Trends today show that HTTP traffic on the Internet is being overtaken by video traffic. But general user increase is still rapidly ongoing. Internet users have grown from 50 million in 1998 to 2.7 Billion in 2013 which is 40 percent of the worlds population. Web sites have grown from 100 000 thousand in 1996 to 670 Million in 2013. [3]

In coherence with this increase the Internets content has also increased dramatically serving more and more heavier dynamic content. With mobile devices also flourishing to be connected to the Internet, mobile users increase traffic even more. This means web servers today play a big part of users life's and they struggle to serve it. This shows the importance of web caches and why they are essential in today's Internet.

2.3 Web server

A web server is an integral part of the Internets World wide web, for clients to surf the Internet there must be a web server present that listens for clients and serves the content back to the clients. Each page request from a client must be generated by the web server using HTML. Since web sites today are very complex containing images, java-scripts, sound, video and more, a browser will request many different aspecpts simultaneously in the background while requesting a single URL. Web servers often contain these images, java-script etc but not always and therefore needs to ask for them from another server before returning the complete HTML web page back to the client. The browsers can also get these from external sites and would be a totally new sequence.

The most commonly used web server software today is called apache though there are others often used today like IIS (Internet information ser-

vices), nginx and GWS (google).

Since web servers have saturation points, web caches becomes a very good solution to this problem and gives a webmaster many benefits. Using a web cache can be an expensive solution if using third party company's, but can be a very cheap solution also tough requiring sysadmin to setup it up themselves. This process can be tedious and hard to do and this is what this thesis is trying to solve, an easier way to manage/setup a open source web cache like Varnish to reduce sysadmin management and make web caching more adoptable for company's.

2.4 Web caching

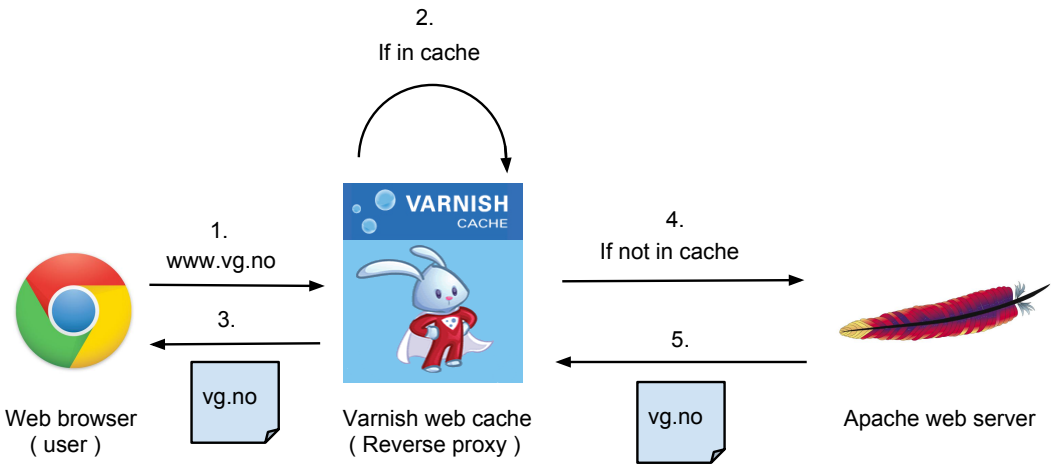
[4] Web caching was introduced to help web servers handle the increasing load on web servers following the huge growth of the World Wide Web. A web cache is put between the users and the web servers taking copy's of responses given by the web servers to the users, so that the next time a user ask for the same page it will receive it from the web cache instead of the original web server. The web cache can store full web pages, images, files etc.

Now there are different ways of caching content. Cookies cache content on the client side via the browser, this helps users when they come back to a site or use the "back" button. It can also help web site remember them so they don't need to log inn again each time they visit.

Proxy caches are caches that users can specify to use via their browsers so they get routed through them instead of directly to the origin server. Proxy caches are also used by the internet service providers on the internet in large scales to help short down requests to different regions. These type of caches are mostly used on the network to reduce latency and network usage. They close the distance from users to web servers over the network.

Reverse proxy caches are mostly used by webmasters that host the original web site to make it more scalable, increase hit rate and be able to handle much more traffic to their web server. Reverse proxy caches gives users the illusion that the requested content came directly from the web server. Reverse proxy caches can yield great performance to websites and are commonly used troughout the world today.

2.4.1 Typical web cache scenario



Typical web cache scenario

1. User request a web page like www.vg.no

2. Varnish checks if the page is in the cache.

3. If the web cache was in the cache, varnish returns it straight to the user.

4. If the web page was not found in the cache, varnish asks the web server for the requested page.

5. Web server returns the requested page to varnish

6. Varnish takes a copy of the web page and returns it to the user [3].

This is the basic functionality of a reverse proxy, this kind of setup is often combined with load balancers, HTTPS proxy servers and more. Tough the core reverse proxy cache setup the same even when combined with other technologies.

2.4.2 Web Caching Algorithms

A web cache is as good as its web caching algorithm, and there are a few of them. Some of these algorithms are quite old but still sufficient today. Algorithms are what make the actual caching work. These algorithms can be used together since they do different things when it comes to a web cache. The different algorithms for web caching are

- LRU (Least recently used) Evicts the item in cache that was not requested for the longest amount of time.
- LFU (Least frequent used) Evicts the item in cache that has been requested the fewest number of times.
- MRU (Most recently used) Evicts the most recently used item first.
- FIFO (First In First Out) Evicts items in the order that they were inserted into the cache.
- Flush (Full cache flush) Evicts all items when the cache fills.
- Random. (Randomly) Evicts randomly selected cache items.
- GreedyDual (Value on each cached item with cost/size.)

The web caching algorithm that's relevant to this thesis would be the LRU since this is whats used by Varnish. This thesis will not go into any depths of this algorithm since it will not be modified/used to help solve our thesis problem.

LRU is an algorithm that simply evicts the least used cached item in a web cache when the cache becomes full. If the cache is not full this algorithm will not be used at all. In coherence's with this thesis we would try to develop something that maybe keeps its own value table of what's important to cache regarding web server traffic logs. If the cache would get full, which it probably would in a real worlds scenario, we could maybe solve this by having a command that updates the cache with our own value table of what to cache. This would mean we would have to measure the amounts of the same type of objects and keep a value list of most used traffic objects and then update the cache regarding these values.

2.5 Other usage of web caching

Other areas web caching can be used is to sort of prevent DDOS attacks. By this we mean that if one would get attacked, there would be huge amount of traffic going to the web server making it crash trying to handle all the web requests. Now using some standard non high end hardware, one can put a Varnish cache for example in front of the web server to automatically drop the most popular requests with an error(700)[5].

Varnish also has a built in feature that automatically drops bogus requests [6].

Since varnish is based on only making a copy of a web page, and only uses shared memory for storage and access, and threads to handle connections it becomes very fast. Its main limitations would be the max amount of memory the caching server has but often its the networks bandwidth that chokes first before varnish does. This is what makes varnish a good DDOS prevention tool.

2.5.1 Web pref-etching

Web pref-etching is somewhat a nice feature that "predicts" which web page a user will access next. If a user accessed for example www.vg.no the web pref-etching web server would then after serving www.vg.no automatically assemble www.vgtv.no so it is ready when the user most likely will access this page since they are very relevant to each other.

The web pref-etching feature will be discussed and maby implemented as a feature later in this thesis.

2.6 Logging

Logging events in this thesis is a crucial part. Logging can be a very useful tool not only for collecting data and events to see whats going on in your system. It can be very effective if one can use this events and information to either influence some part of the system or using it by another program to either calculate, reconfigure or information to see what's really going on. Logging in this thesis will be important since it will be logging the web server traffic coming from the Internet, the information going from the varnish web cache to the web server will be crucial part when deciding what the varnish web cache should cache.

2.6.1 Common log format

The common log format is a standardisation of logging that's being used by almost all web servers today. This makes it very easy for other software to read these log files. Having a standard log format saves time for system administrators to understand the logs and also much easier to use/manipulate them for other means and usage.

Example

The structure is as followed:

```
host ident authuser date request status bytes
```

Having this standardisation makes it much easier to retrive specific information and also helps make it mor readable.

2.7 Varnish

Varnish is an Open source HTTP-accelerator reverse proxy web cache soultion. It was developed by a Norwegian online newspaper called Verdens Gang (VG). What separates varnish from other web caches is that its designed from the ground up to do one thing, accelerate HTTP traffic only. Many other web caches support FTP, SMTP caching and were not originally meant to be used as a reverse proxy but as client side caches for example. This is what makes the Varnish so good, its made from scratch with one purpose. This means all aspects of it are well tought trough that should make it fast, easier to use, easier to implemenent, easier to install and to modify then other web caches. Its optimized for 64-bit, multiple cores, meaning its designed with the future in mind.

Varnish is also very scalable by how its architecture is. Threads and virtual memory storage are what makes the varnish so fast, meaning it doesn't need to wait for disk writes or reads if used by other parts of the system. Varnish's only real performance limitation is the speed of the network. Varnish also supports plug-ins, IP based directing and Gzip decompression and compression. A nice feature that varnish also has is VCL (Varnish configuration language), its used to define request handling and document caching policies, by using a VCL its much easier to configure the varnish web cache. It gives users the ability to tune, tweak and override the default varnish mechanics "under the hood".

Another feature with varnish is that as soon as you install it can just add what back-ends you want to cache, it will start working. That means it will automatically cache all GET requests. This is quite a nice feature, where one can later use the VCL to configure it to cache whatever one wants.

2.7.1 HTTPerf

HTTPerf is a tool created by Hewlett Packard for measuring web server performance. HTTPerf is a good tool for testing different aspects of a web server performance. For this thesis it will be used with a combination of unique urls to simulate a controlled amount of traffic to the web server, but also simulate "real life" traffic later in the experiment. This will help to find out how varnish reacts to different traffic loads and how it handles unique and duplicate urls.

2.8 Automation

Automation is something that is very much used by network and system administrators today to save time, resources and reduce configuration and human errors. Automation is greatly used where one has a job or process that needs to be executed several times. A best practice for a good network

and system administrator is that if one can automate a process, then do it [7].

Automating aspects of systems can really save time and help reduce sysadmins workload. Automation is a key aspect for network and system administrators since they often are involved in many aspects of the company's IT system. The concept of automation can be achieved in different ways, one can use tools, script and schedule's to achieve automation. When using own methods to achieve automation they become an unique solution, documenting your solution is important for own usage but also so others can implement, use and improve your unique solution.[8]

2.9 Best practice

Technology solutions today can be quite complex and unique and standardisation is not always used but a goal for all technologies but hard to archive since so many technologies make up a solutions. Since so many technologies can be used to achieve a task or solution one needs to rather focus on the bigger picture then the unique technologies.

This is where best practice comes in. Best practice means to follow a standard way or thinking to describe a process that can improve and evolve. Its a way of maintaining quality when difficult, new or unique solutions lack standardisation. Best practice has become a big part of business today and a network and system administrator can use best practise methods and templates with their own interpretations as guidelines to accomplish their own unique solution. [9]

Since many solutions today are based on best practice, this thesis will explore a best practice way to make web caches more manageable, easier to implement and in turn create a sort of template or standard that system administrators wanting to implement web caches can use.

2.10 Machine learning techniques

Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data. [10]

There are many Machine learning techniques (Learn from data) today that can be used for different results and purpose. What algorithm to use where, all depends on what data one has and what kind of information/statistics or results one wants to collect.

The machine learning technique relevant to this thesis would be ranking algorithms.

2.10.1 Ranking algorithms

In this paper a ranking algorithm will be made for ranking URL's comming to the webserver using perl scripts. The reason for this is that one needs to separate the more frequently used urls beeing requested to the web server which will then be made into a web cache rule. This would be a "learn from data" approach where one ranks URLs based on frequency.

2.11 Related work

This thesis involves a few areas regarding network and system administration, web caching and computer science. This thesis covers some aspects of the network and system administration proficient, where some related topics influences this paper. Related work around this thesis problem statement do exist but covers more specific aspects and some generalised work that becomes to vague to be related to this thesis.

Chapter 3

Model Methodology and Approach

System administrators who want to implement a web cache solution will have to explore different methods of implementing the solution. Different factors come to mind here, solutions available, documentation available, training needed, cost, implementation time and management. Taking in all these factors a system administrators will consider a solution best suited for their company's needs.

Varnish would be a good choice regarding all factors previously mentioned. Varnish is open source, free, it has good documentation but some training is needed, implementation time is needed and management. Now if training and implementation time could be greatly reduced, Varnish in coherence with this master thesis solution would be a very good choice when wanting to implement a web cache solution, and also ease adoption of web cache solutions in general making web caching more convenient and popular.

This chapter will cover the design, terminology used and functionality of the suggested design. Followed up with a suggested prototype that will be used for testing and completing an experiment which will try to explain and show trough the analysis if the problem statement can be achieved.

3.0.1 Alternative approaches

A simpler but not as relevant way to solve the problem statement would be to just do simulations rather then making a prototype and do experiments to find a solution.

These simulations could be done using R-studio where one would simulate web traffic and caching. By this meaning more to show how web caching works in general using simulations, showing how a web cache would behave when using caching rules. Using simulations can also somewhat show how the idea of having a method of ranking most popular urls on the

web server and then implementing them into the cache automatically. Using this method could show how traffic would behave. This would eliminate the varnish learning curve needed to understand how exactly varnish works but how web caching works in general.

3.1 Approaching the problem statement

Understanding the problem statement by breaking down each key word will help get a better understand what this thesis is trying to solve. But also use other keywords to help describe underlying ideas.

Automated web-cache **configuration** using **machine learning**

Process
How one configures a web cache.
What actions/steps?

Configuration
What needs to be configured in a web cache?
What is needed for a web cache to work?

Automated
How can it be automated?

Machine learning
What machine learning can be used?
How one can make automate configurations in coherence with machine decision based on properties.

Each of these aspects are areas this thesis is going to try to solve using different methods in the approach.

3.1.1 Process

Web caches (reverse proxy's) share the same goal and idea behind their configurations but differ in the way they are configured. They all require configurations to work but are in no way similar to configure.

The web cache used in this paper will be Varnish. Varnish has its own way of being configured using their own language called VCL [11] as described in the background chapter 2.7

The process needs to be mostly automated using some sort of ranking system to know what URLs needed to be used in the web caching rules.

3.1.2 Configuration

How does one make caching rules and how they are aapplied? Solving this part and reading up on how the varnish works in general in combination with how their web caching rules are written and applied. Installing a varnish web cache and writing simple caching rules combined with some simple tests will help figure out how all this works.

3.1.3 Automated

How to automate the configuration proccess can be solved by first knowing how varnish applies its caching rules. Then making a script or function in a script that does this part automaticly.

3.1.4 Machine learning

Research on how different machine learning techniques work will help figure out which one is relevant to this thesis. After knowing what technique one can use, one can start making an own version using one or more scripts.

3.1.5 Web server logging

Web server logging on an Ubuntu server uses the common log format which makes it very good to analyse with other software. It also makes it easier to make a script that's going to use this information. The web server will be logging all HTTP traffic, by this we mean GET, POST etc. This information will then be used to see what traffic is going to the web server.

3.2 Design

Having a design helps understand and see the big picture of this thesis and outline how one would approach such a problem statement. Different methods will be suggested with ideas on how this approach will be outlined.

General models will be used to help visual aspects of the approach, this will give a better insight and overview of what really going on. It will also try to simplify some general aspects for better understanding different aspects of the approach.

Using design and models can be used as references when making scripts for different aspects of the approach. Having a design of what one is trying to make, helps having an overview when making each aspect. It can benefit the reader by referring a model to different part of the text.

Pseudocode [12] is another method that's going to be used to help describe code in a simpler and more readable form.

3.3 Algorithm concept

3.3.1 Model and Terminology

The models will include terminology, descriptions, event flow and general properties. Using graphical models will help get an easier understanding of how the prototypes will be work.

First a basic model will be used to show a more bigger picture of what this thesis is trying to solve.

The second model will describe what kind of proposed algorithm is going to be used for automating the server side.

The third model will show the ranking algorithm proposed using machine learning techniques for establishing policy rules that are going to be implemented into varnish.

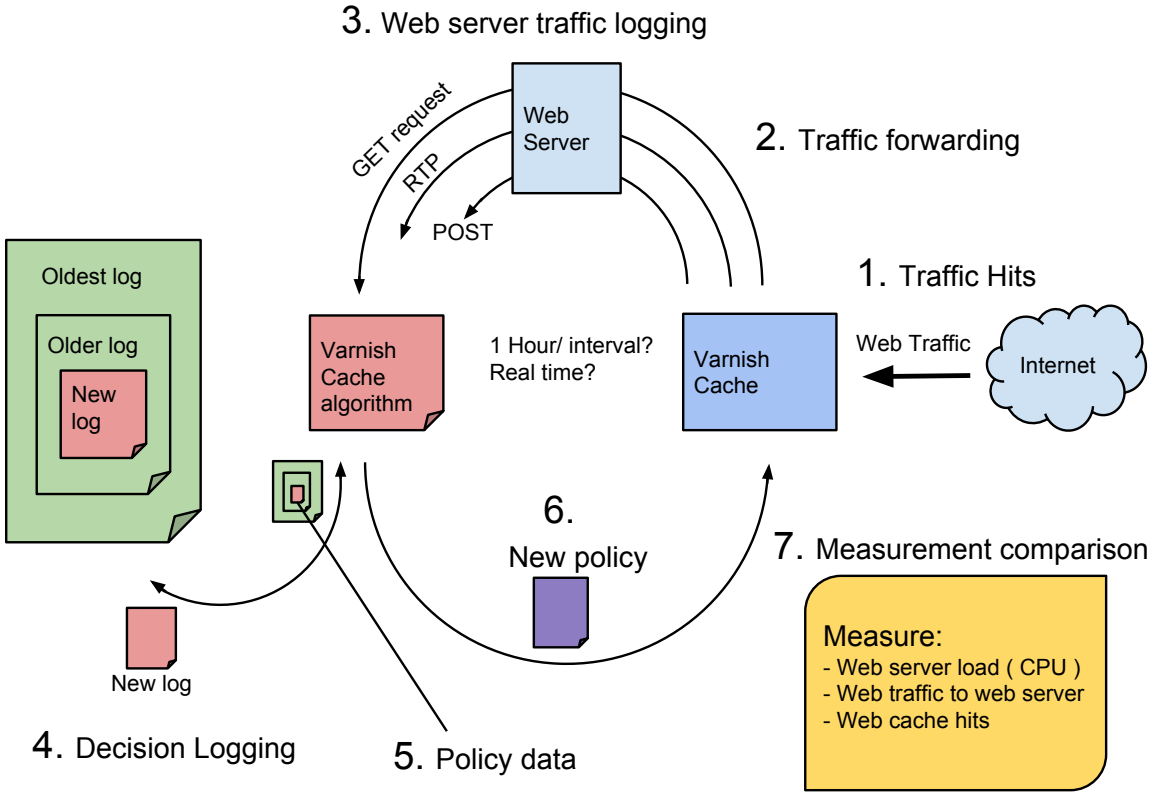


Figure 3.1: Algorithm concept

Model explained

1. Internet traffic hits the "front side" of the Varnish web cache.

2. Traffic that is not cached will be asked for by the varnish cache from the web server.

3. Monitoring the web server access.log will show what request are being asked for.

4. New urls that hit the web server will be added and update the "main" log file.

5. Ranking algorithms are run on the main web server log.

6. Policy is written in varnish language (VCL) and implemented in the varnish cache flawlessly while varnish is running.

7. Compare new web server load statistics with old measure meant to see if new urls are being cached.

Compare new web server traffic with old web server traffic to see a decrease.

Compare web cache hits with old web cache hits statistics to see an increase in hits.

3.3.2 A day in the life of a URL

Important URL vs Non important URL.

3.4 Prototype

3.4.1 Default.vcl

To approach our problem in this thesis a look at varnish's own web caching rules needs to taking into consideration. The default.vcl is a file that varnish needs to have present even if one makes its own web caching rules. This file helps varnish behave in a secure and default manner after ones own rules is catch-ed and applied. The default.vcl file can be used to make general rules that decide what to cache, and a common rule [13] that can include different file types like .jpg, .css etc and use an OR (|) value to separator them so that the web cache checks if whats hitting the web cache is either of these file types.

This helps varnish cache static content that's used a lot but the rule becomes to "general" and inconsistent to really work well.

To make varnish learn what to cache we need a few things.

- 1. A web server traffic log that shows what HTTP url traffic is being used.
- 2. A script that reads these log files and interprets them in the right manner so the right information is being used.
- 3. A script to keep old log files and add only new http url traffic.
- 4. A script that takes http url's and makes varnish policy rules from them.
- 5. A script that implements these policy's into the varnish web cache.
- 6. A measurement comparison of the web server load (CPU) before and after policy implementation.
- 7 A measurement comparison of web traffic to the web server before and after policy implementation.
- 8. A measurement comparison of web cache hits before and after policy implementation.

3.5 Testing/Experiment

A measurement comparison will show us if these new policy's in varnish are being used by seeing if the web cache hits increase and the web server traffic and load decreases.

A combination of diagrams with text will be used to display the experiments success.

To measure success in these experiments 4 variables will be measured and logged. After each experiment a comparison and coherence between these 4 variables will be analysed.

HTTPerf will help controll this behavior of varnish by using a specific amount of urls a limited amount of times.

HTTPerf [14] will be used to generate consistent equal amount of traffic during the experiment to ensure a fair and genuine result.

With HTTPerf one can control exactly how much traffic one wants to send and how often. For the experiments a combination of two HTTPerf will be used to simulate regular normal steady traffic and spike traffic.

3.5.1 Experiment overview

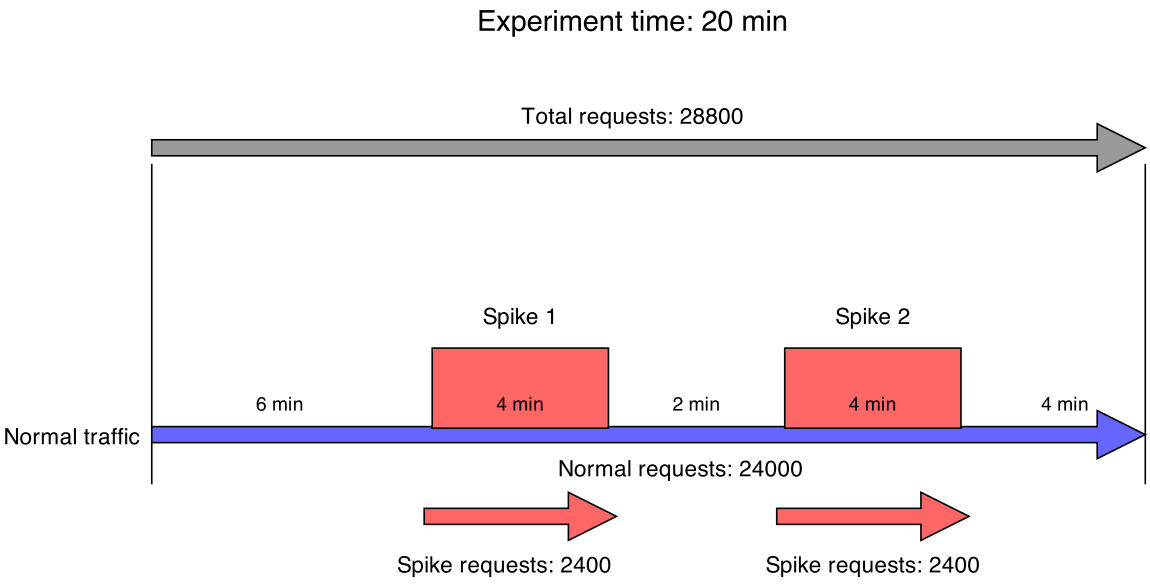


Figure 3.2: Experiment overview

3.5.2 Normal traffic

The normal traffic will be run using a script with the following HTTPerf command:

```
#!/usr/bin/perl

# Normal traffic. 10 min test
$varnish_long = system( "httperf --server 10.0.4.2 --wlog Y,
wlog_LONGx100_uniq.log --num-conns=12000 --rate 20
--num-calls 1 --verbose");
```

This command will request 12000 urls with a rate of 20. This will result into a 10 min test.

3.5.3 Spike traffic

The Spike traffic will be run using using a script with the following HTTPerf command:

```
#!/usr/bin/perl
sleep 180;
```

```
#  Spike.  2 min test
$varnish_short = system("httperf --server 10.0.4.2 --wlog Y,
wlog_SPIKEx20_uniq.log --num-conns=1200 --rate 10
--num-calls 1 --verbose");

sleep 60;

$varnish_short2 = system("httperf --server 10.0.4.2 --wlog Y,
wlog_SPIKEx20_uniq.log --num-conns=1200 --rate 10
--num-calls 1 --verbose");
```

This script will first sleep for 3 min (180 sec) before running the spike traffic. This traffic will run for 2 min.

Then the script will sleep for 1 min (60 sec).

After 1 min the spike traffic will run for 2 min.

3.5.4 Experiment Traffic flow

100 unique urls will be used for the **normal traffic**, and these 100 will be requested 12000 times. This will give a nice steady traffic that will help proof the concept of the script being run.

To simulate somewhat real world scenarios where traffic spikes very often occurs the spike script will request 20 unique urls using **normal traffic** urls. This is to simulate a spike on 20 selected urls.

By using these two scripts simultaneously a nice flow of traffic will run for 3 min, then a extra spike for 2 min, followed by 1 min of **normal traffic** flow and again a spike running for 2 min, ending with 2 min of **normal traffic** flow.

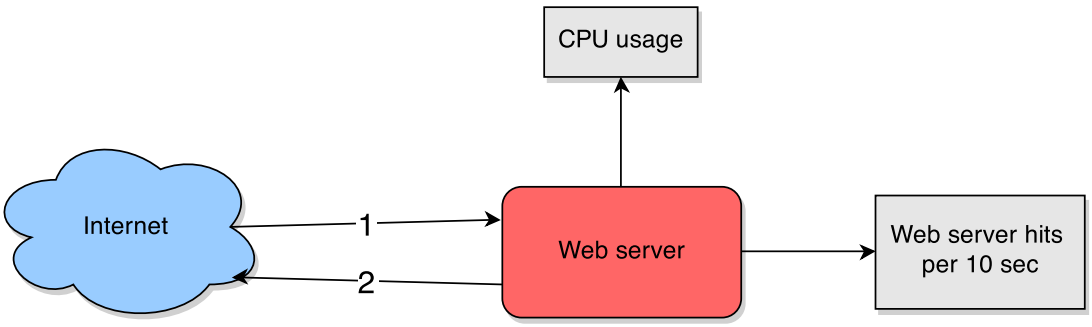
This will in the end results in a 10 min test with a total 14400 unique url requested.

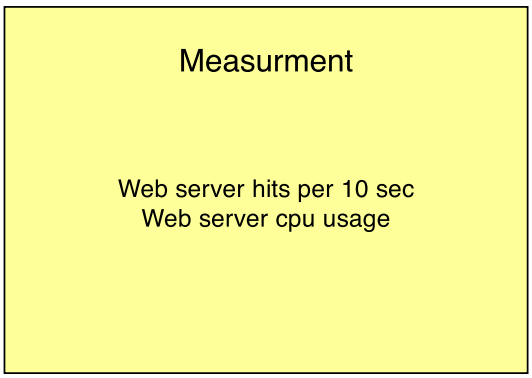
3.5.5 Baseline test

A baseline test will first be performed to have a comparison to the actual experiments, this will help give proof to the experiment and to see how much variation the scripts give.

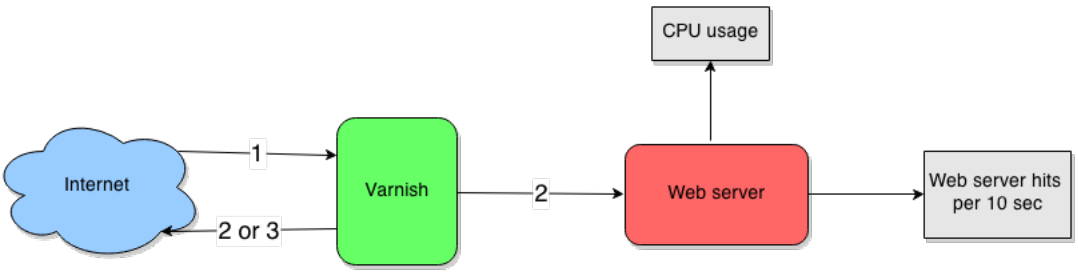
This means running two experiment, one against the web server and one against varnish without the proposed ranking scripts.

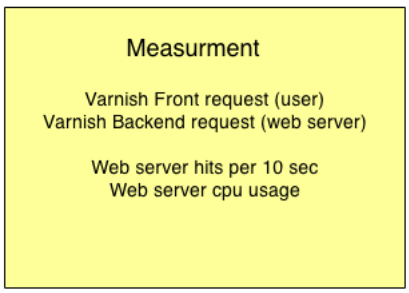
3.5.6 Baseline test design against web server





3.5.7 Baseline test design against varnish





On the varnish server two variables will be measured using two varnish commands:

Front end (User hits)

```
varnishtop -i rxurl

This command will show how many hits the users side of varnish gets.

Success = High hits
```

Back end (Web server hits)

```
varnishtop -i txurl

This command will show how often the varnish requests the web server for a url.

Success = Low hits
```

These two variables will show how well the new cache rules apply.
The goal here is that varnish should return as many request as possible to the users and not ask the web server for the url.

3.5.8 Web server measurement

Web server hits will show how much traffic request are going to the web server. This will be measured each 10 sec trough the whole experiment to get a better understanding on how effective the script is trough the whole experiment.

```
while true; do echo $(date +%s)  $(wc -l /var/log/apache2/access.log |
cut -f 1 -d " ") >> baseline.dat ; sleep 10; done
```

3.5.9 Expected results

A measurement comparison will show us if these new policy's in varnish are being used by seeing if the web cache hits increase and the web server traffic and load decreases.

Success scenario

A success scenario will have a decrease in requests from Varnish to the web server and an increase in Varnish cache hits. A success would also be if caching rules are updated in the varnish cache, a simple view of the varnish rules will prove this.

Chapter 4

Results

Different results aspacts from the approach will be presented with a combination of code and graphs to best illustrate each aspect. The results chapter will display code used to make each algorithm combined with an explaination. Further graphs will show the outcome of each tests performed.

4.1 Web server URLS

The first thing needed would be to get URLS from the web server log. URLS will be gathered using their timestamps, this is to know what urls have been gathered each time. This means that when say a log file has 100 urls all of them will be gathered first, as time passes and traffic to the web server is somewhat consistent new urls will appear.

So the second time there will maby be 150 urls in the log file, only the new 50 urls will be gathered which will guarantee only new URLS.
This key part will be a basic function in all 3 algorithms suggested. Since all 3 algorithms would need urls for ranking.

```
#### Web server URLS ####

 $U_time = current time;

 save $U_time to file;

 $Last_U_time = $U_time from file;

 # Read LOG timestamps

 Open LOG access.log.
```

```
While each line in LOG
   {
       if ( $line matches timestamp regex )

           # convert timestamp to EPOCHE time, for comparison.

               my $O_time = EPOCHE timestamp($line);

               if ( $last_U_time < $O_time )

                   {
                       while ( each $line2 in access.log )
                           {
                               if ( $line2 matches URL )

                                   {
                                       Put urls i hash;
                                       Count total URL;

                                   }
                           }
                   }
   }

save $U_time to file;

# Start ranking scripts
```

4.1.1 Proposed ranking algorithms

3 ranking algorithms will be proposed.

The first ranking design will try to choose URLS's based on the frequency amount they show up in our URL list, and have a linear lifetime during each interval meaning they will only exist in our list for so long. The popular the URLS will stay longer in the ranking list.

4.1.2 Design 1. FBRLA

The first design will be the most basic method of ranking URL's wich will in the end be inserted into the varnish's .vcl language.

This first design will be called **Frequency based ranking with linear attrition** (FBRLA).

Frequency is how often a web page is requested by varnish to the web server.

Based ranking means ranking based on frequency. Which in terms means ranking based on how often something occurs.

Linear attrition is when something decreases in a consistent rate.

Below is the proposed ranking script achieving Frequency based ranking with linear attrition.

```
###### FBRLA Ranking script psudo code ######

### FBRLA START ###

$N = 20;   # Max URLs in %URLhash
$AT = 2;  # Attrition value
$ET = 4;  # Eviction Threshold

### Read P file URLS and put into $P array ###

Open P.txt

While ( each line in P )
    {
        Split each line with "/:/";
        Put URL with AT value ( mana ) into $P{$url};
    }
close file;

## Importance ##

for all items in %P;

{
  reduce all keys with AT(10);
}

## Eviction Threshold ##

$evicted = 0;
```

```
    if ( keys in P are larger or equal to Max size ( $N )
        {
            foreach ( keys in P)
            {
                if ( URL is smaller then Eviction threshold
                ( $ET )
                {
                    delete URL;
                    Update total evicted;
                }
            }
        }
    } else
     {
        print " Not evicting, P not large enough: " Total keys
        in $P ;
     }



## Add new URLS ##

$added urls = 0;

foreach URL ( sort keys ascending )

    {
        if ( Total keys in %P are smaller then Max size ( $N )

        {
            URLs in P = New urls / total urls;
            $added urls ++;
        }
    }

## Save P to file ##

Open P.txt;

foreach $key ( in %P )
    {
        write URLS from %P to file P.txt;
    }
close P.txt
```

```
### End of FBRLA ###



######## Add urls into Varnish rules #########

### Make .vcl file to be used in varnish ###

## Start of the rule ##
$midstart = Varnish caching rule;

## URLS to be added into caching rule ##
foreach URL in %P
    {
        midstring = URL;
    }

$midend = )) ;

Combine $midstart, $midstring and $midend into one string
$midstring;


Open mid.vcl file and write $midstring to file;

concatenate 3 .vcl files ( head.vcl, mid.vcl tail.vcl )
into new.vcl;


### Secure copy new.vcl to varnish server ###

scp new.vcl to varnish;


## Start a script on varnish that uses the new.vcl inline
( no varnish restart required )

ssh varnish /telnetvarnish.sh




####  Telnetvarnish.sh ####

Connects to varnish administration console via telnet;
```

```
Use default.vcl;

Discard current new.vcl;

Load new vcl ( FBRLA ) new.vcl;

Use new vcl ( FBRLA );


###### END OF SCRIPT ######
```

After this script is run, varnish will now have a new .vcl config that will be in use while varnish is running, meaning no service restart or reboot is needed for it to be used. This is a great advantage in a production environment, meaning the varnish service will always be up and running.

Proof of concept (config)

```
root@varnish:/home/ubuntu# varnishadm -S /etc/varnish/secret
-T localhost:6082
200
-----------------------------
Varnish Cache CLI 1.0
-----------------------------
Linux,3.2.0-57-virtual,x86_64,-smalloc,-smalloc,-hcritbit
varnish-3.0.5 revision 1a89b1f

Type 'help' for command list.
Type 'quit' to close CLI session.

varnish> vcl.list
200
available       1 boot
active          2 FBRLA
```

Under the vcl.list command one can see that the original default boot config is not in use, where the FBRLA config is now in use.

Proof of concept 2. (caching rules)

```
varnish> vcl.show FBRLA

# CACHING RULES START

sub vcl_recv {

if (req.request == "GET" && (req.url == "/normal/normal1.html" ||
                             req.url == "/spike/normal2.html" )) {
    unset req.http.cookie;
        unset req.http.Authorization;
  return(lookup);
  }
else
{
   return ( pass );
}
  }

  sub vcl_fetch
{
  return ( deliver );
  }
```

Here we can see the two rules generated by the FBRLA script which will be cached in varnish.

The **AT**(Attrition value) is a value that can be adjusted to tweak the ranking.

Higher AT = Faster URL replacement.
Lower AT = Slower URL replacement.

This value by default has been set to 10, but can be adjusted for better coherence with traffic going to the web server.

The first algorithm is not optimal but is more used to prove the concept of ranking urls.

4.1.3 Design 2. TRRLA

The second design will build on the first design, it will include a new function called Temporal recurrence.

This design will be called **Temporal recurrence ranking with linear attrition** (TRRLA)

Temporal recurrence ranking means to rank something based on how long it existed based on the first observation and how often it occurs again. ??

is when a object exists for a limited time, but how often it occurs again for since it last existed.

Linear attrition is when something decreases in a consistent rate, in this case the attrition value of URLS ranked.

Psudo code
The TRRLA script will be very similar to FBRLA, only now adding a few new functions to the eviction part of the script.

```
###### TRRLA Ranking script psudo code ######

### TRRLA START ###

$N = 20;   # Max URLs in %URLhash
$AT = 2;  # Attrition value
$ET = 4;  # Eviction Threshold

### Read P file URLS and put into $P array ###

Open P.txt

While ( each line in P )
    {
        Split each line with "/:/";
        Put URL with AT value ( mana ) into $P{$url};
    }
close file;

## Importance ##

for all items in %P;

{
  reduce all keys with AT(10);
}



## Eviction Threshold ##
```

```
open (Evicted_time_url.txt)

$evicted = 0;

    if ( keys in P are larger or equal to Max size ( $N )
        {
            foreach ( keys in P)
            {
                if ( URL is smaller then Eviction threshold
                ( $ET )
                {
                    $evicted_time = time;
                    print $evicted_time and evicted_url to
                    Evicted_time_url.txt;

                    delete URL;
                    Update total evicted;
                }
            }
        }
    } else
     {
        print " Not evicting, P not large enough: " Total keys
        in $P ;
     }

close (Evicted_time_url.txt)


## Add new URLS ##

my $added = 0;

for each $url ( sort ascending )
{
    # grep url from evicted file
     if ( total keys in %P < $N )
     {
        my $EvictedURLs = get last evicted url from evicted
        file;

        # Get last evicted url TIMESTAMP
        if ( $EvictedURLs )
            {
                @EvictedARRAY = split content into array with
                "," to get timestamp;
```

```
                my $interval = (( Evicted time - url timestamp
                / 20 );

                # New mana

                my $newmana = Url frequency / total urls
                * 1000 );
                $newmana = (Url frequency / total urls *
                1000) * ( 1/$interval)
                if $interval > 0;

                $P{$url} = $newmana;
            }
             else
             { $P{$url} = standard mana 10;
        }
        $added ++;

    }

}


## Save P to file ##

Open P.txt;

foreach $key ( in %P )
    {
        write URLS from %P to file P.txt;
    }
close P.txt


### End of FBRLA ###
```

4.1.4 Design 3. FBRTAA

The third design will be a combination of the first and second algorithm which will get the best from both. It will include the general ranking from the first design, temporal recurrence from the second design and include a new function called temporal attrition amplification.

This design will be called **Frequency based ranking with temporal attrition amplification**. (FBRTAA)

Frequency is how often a web page is requested by varnish to the web server.

Based ranking means ranking based on frequency. Which in terms means ranking based on how often something occurs.

Temporal values lasting only for a time, only in each iteration. This will regulate the attrition value to correspond to the specific amount of traffic going to the web server.
Attrition amplification means how much should the attrition value for each url be amplified regarding the interval.

4.2 Experiment data

4.2.1 Baseline experiment

Each test was controlled to the same identical test parameters regarding HTTPerf traffic.

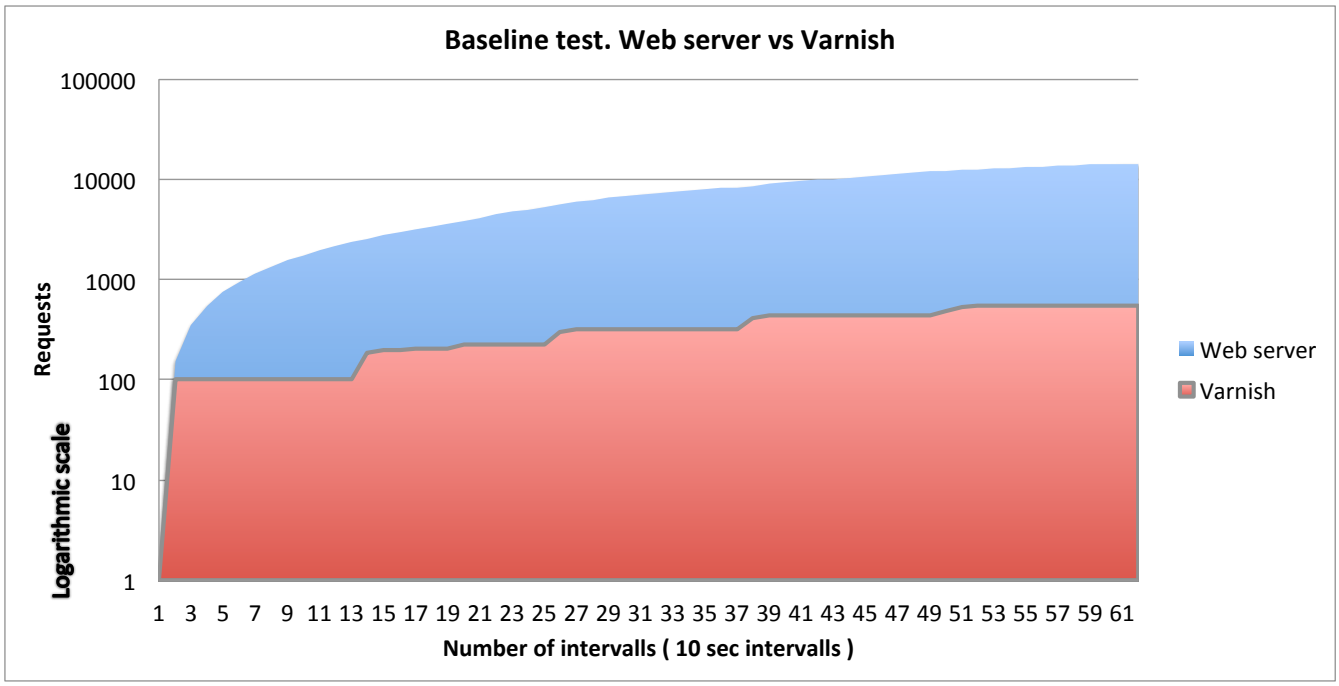


Figure 4.1: HTTPerf to webserver vs varnish

In the 4.1 figure we can see the web server getting all 14400 request vs when going trough varnish the web server only got around 540 requests.

This means varnish by default cached 96 percent of all the traffic.

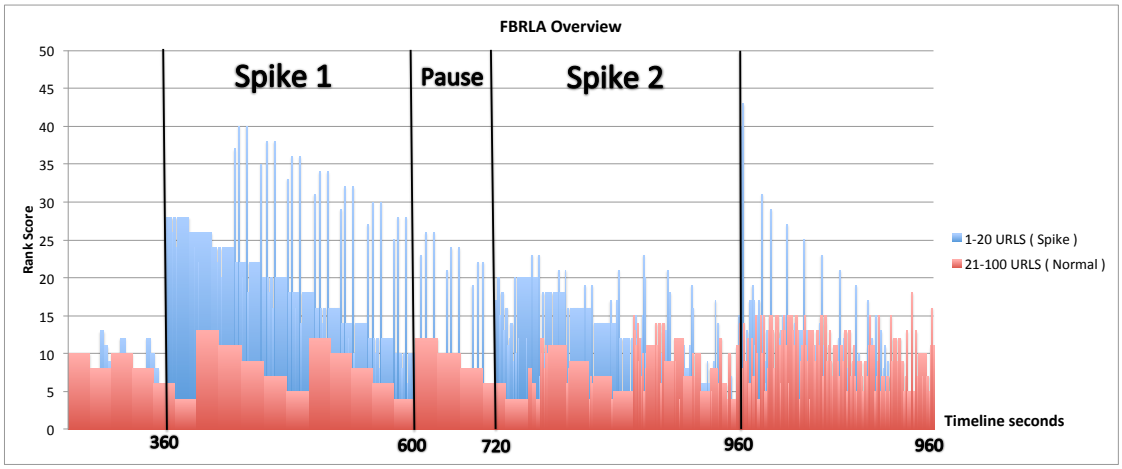


Figure 4.2: FBRLA

Figure 4.2 shows FBRLA ranking urls in a linear way. Before the first spike of traffic the FBRLA algorithm ranks normal traffic consistently.

During the first spike we can se it ranks the spike traffic much more.
We can clearly see the spike traffic pausing, but normal traffic still being ranked in a linear fashion.

When spike 2 starts the FBRLA algorithm ranks the spike traffic a bit more then the normal traffic.

Since the first and second spike are identical in their properties regarding amount and run time, we can see the aftermath after spike 2 resulting in a spike tail lasting the same amount of time as spike 2.

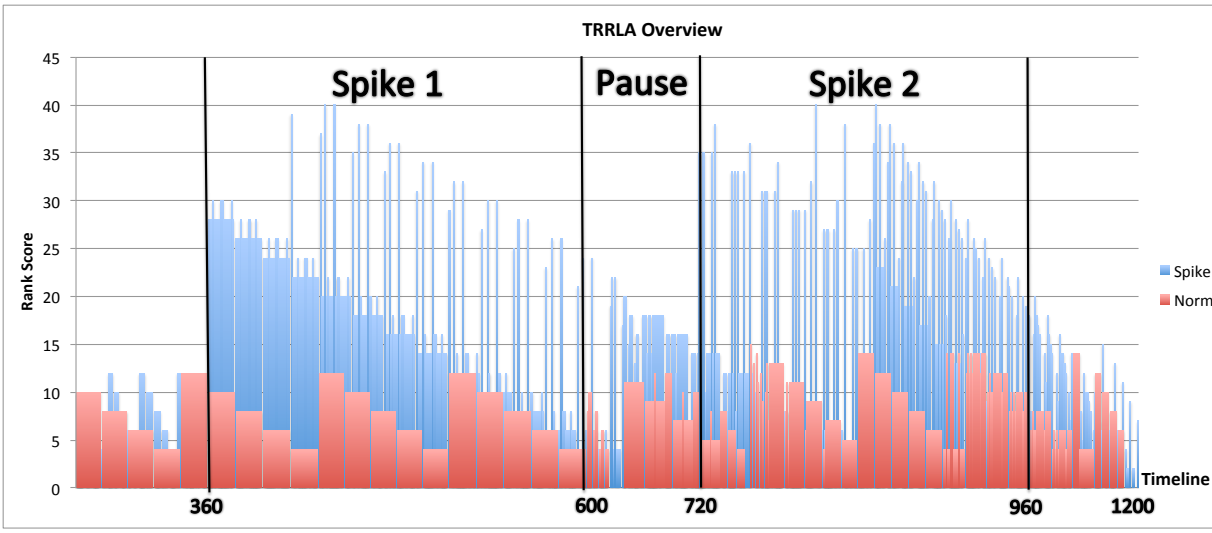


Figure 4.3: TRRLA

Figure 4.3 shows the TRRLA ranking urls in a linear way. Before the first spike of traffic the TRRLA algorithm ranks normal traffic consistently.

During the first spike we see that it first starts to rank normal traffic and soon after the spike traffic. We also see that it ranks recurring urls higher.

When the spike traffic pauses we can see TRRLA still ranking normal traffic in a linear fashion.

During spike 2 we can see recurring urls from the first spike being ranked higher followed by the normal recurring traffic.

As like the FBRLA algorithm the TRRLA algorithm also creates a spike tail of recurring urls both spike and normal traffic diminishing linearly. Tough spike traffic being ranked higher in general.

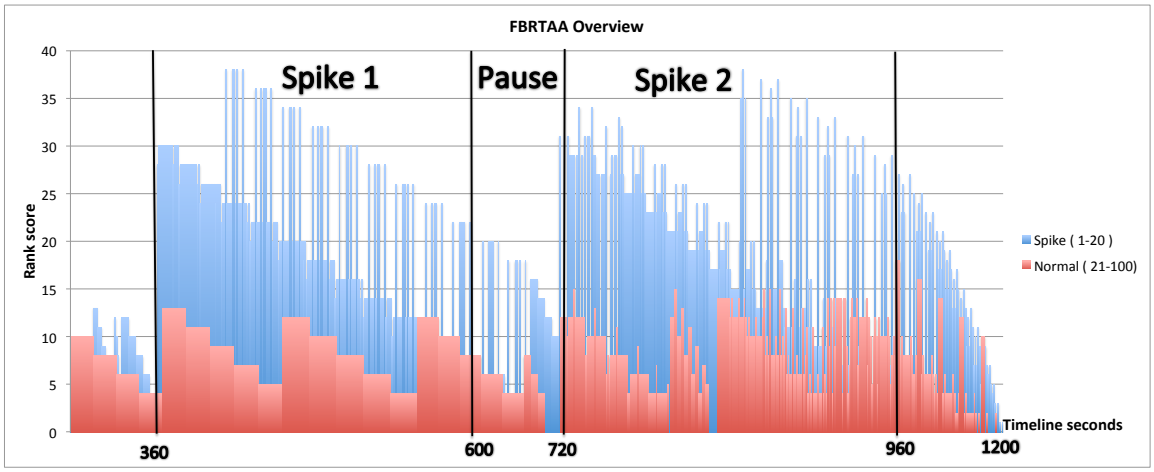


Figure 4.4: FBRTAA

Figure 4.4 shows the FBRTAA ranking urls in a linear way. Before the first spike of traffic the FBRTAA ranks normal traffic consistently.

During the first spike we can see that it starts to rank the spike traffic much more then the normal traffic in a linear way.

Then there is a pause of 120 sec.

Followed by the pause is the second spike ranking urls with the new Attrition amplification. We can see this by the greater ammount of spiked traffic rank and grouping.

As like the two previous algorithms it also creates a spike tail of reccuring urls, both spike and normal tough less normal traffic.

Chapter 5

Analysis

5.1 Analysing the results

Analysing the results will give a bigger picture understanding to see how each script behaves and works throughout the experiment.

5.1.1 Interpreting the results

Each algorithm do rank spiked data higher then normal which is what they are supposed to do. Tough clear differences can be seen. Lets evaluate each of them to really know whats going on. But first lets look what they have in common.

1 Interval = 20 sec.

Similarity's

All 3 graphs shows a combined traffic of both 1-20 urls and the 21-100 urls for 360 sec. All 3 are quite the same tough showing the normal 21-100 urls being slightly less ranked before the 360 sec. This shows each algorithms own feature at work.

Spike 1
At the first spike we can see all 3 algorithms start prioritising the spike traffic with about the same rank (27-30).

Slightly after we see all 3 use about 60 sec (3 intervals) before singling out specific spike urls whom gets an even higher rank. This happens because the size of N has a limit of 20, meaning only 20 can "live" in the algorithms during each interval.

Linear attrition All 3 graphs show a linear attrition during the first spike.

FBRLA

The first figure 4.2 shows that it performs Frequency based ranking with linear attrition.

Spike 1 shows a steady linear attrition even in the paused part.
Spike 2 shows the algorithms starts to rank spike traffic more tough not as good as spike 1. The reason behind this could be that since it gives so many URLS in spike 1 so high rank, they don't have time to be evicted.

After **Spike 2** a small peak of spike urls occur with a narrower slope meaning a higher attrition value is used. One can also see that it starts to cache the normal traffic as much as the spike urls. This is expected since normal traffic consists of both spiked urls and normal urls. This simply shows the algorithm doing what it should be doing.

TRRLA

The second figure 4.3 shows alot of the same behavior as the FBRLAA but a clear different is what happens when **Spike 2** occurs. What happens here is that the urls from **Spike 2** which are the same as **Spike 1** occurs again, they now should be given a greater ranked if they appear again. The point here is that if an URLs shows up again after a short while, it should be considered more important then the rest.

Spike 2 also shows mid spike inside it which occurs beacause the algorithm sees the same urls again now showing up even faster and considers them more important by giving them a better rank again.

A pattern here would be that the often it occurs again the more important it will be, in return lasting longer in the cache.

FBRTAA

The third figure 4.4 shows some of the same behavior as the previews algorithms tough with its on features.

The differences can be seen starting at the first spike **Spike 1**.
In **Spike 1** one can see that it gives more urls a ranked score then the TRRLA.
Unclear why this is.

After the pause we can see the textbfSpike 2 giving about the same ranked score as TRRLA tough much more consistently with more urls.

Whats interesting with the FBRTAA is what happends midway trough **Spike 2**, often occuring urls are divied by their interval time since last occured meaning the will be given less rank.

At the end we see the important urls get a lower rank faster as they should.

5.1.2 What have we achieved

This paper has eliminated the need for manually updating what a web cache should cache.

This thesis has made an algorithm for deciding what URL's should be cached based on their importance.

5.1.3 Automated caching

Automated caching is achieved using the telnetvarnish.sh scripts in combination with the 3 proposed algorithms.

5.1.4 Machine learning techniques

Machine learning techniques have been used to optimally decide what urls should be used for caching.

5.1.5 Management reduction

Automating web cache rules is a process that needs 2 different methods to achieve.

Algorithm script which finds what URLs that should be used in the web cache.
A**script on varnish** that updates its own default.vcl with caching rules via telnet.

By using the FBRTAA in a combination with the telnetvarnish.sh script one does automate what URLs to cache should be cached and the need for not manually writing caching rules in the default.vcl.

Chapter 6

Discussion

This chapter will discuss each aspect of the thesis regarding approach, results and overall process.

6.0.6 Results

The results showed us a clear difference of each algorithm proposed in this thesis. In a combination with the experiment proposed the graphs helped show how each algorithm ranked traffic. The experiment also help prove that each algorithm actually worked the way they were intended.
The results were quite reliable and not unexpected.
Using each algorithm on other web caches then varnish with some modifications could help show how universal these algorithms are. Since they in theory could be used on any web caching technology.
Reproducing the results would be possible using the same tecqhniques used in this thesis.

6.0.7 Process

The project started out as a very good idea and somewhat ambitious.
This resulted in many ideas on how one could answer the problem statement.

The first obstacle encountered was to figure out what kind of machine learning technique that would be optimal for this thesis. This required a lot of time to figure out and could have been prioritesed more.

Another obstacle that consumed a lot of time was figuring out how varnish's default logic worked. This created an obsticle for completing the experiments tough a small workaround was used to make the experiments work.

Due to time constraints testing the proposed algorithms on other web cache systems was not possible. Testing it on other web caches could have helped improve the algorithms and also really showed not only in theory

that they could work on other web caches.

An alternative approach was proposed that could have made it easier and given more insight into how a ranking algorithm could have been programmed.
More experiments could have been performed using different data measurments which could have maby have given a clearer view of how each algorithm performed and worked. Due to time constraints this was not possible.

The problem statement was altered but didnt really change the approach in any way.
Things that could have been done different would be more research around web caching and even more around machine learning.

6.0.8 Problem domain and impact

Varnish vs our approach

Varnish itself is quite powerful, where it can be run by default only specifiyng a web server backend and perform very well caching upto 95 percent of all traffic.
This is because of its own default built in logic which can be overridden if needed.

Tough its not ideal to run varnish this way since it needs to be tweaked to comply with the companies specific system and needs. There are many sites one doesnt want varnish to cache for example. And one often want it to behave in other ways specified by each companys requirments.

So the main difference between running varnish by default and using the propopsed algorithms would be that the algorithsm would find out what kind of traffic one should priorites to cache, not just caching everything.

Web caches

The algorithms in this thesis are programmed to run against a varnish web cache, but since the main part of the algorithm gets its information from the the apache2 log file, which uses the standard log format. This means the algorithm can be used on other web caches. Tough it would need to be modified to be used with the other web cache, since the main part of the script finds what urls should be cached, but another function writes them to the web cache.

This could easily be achieved with some basic script functions and small modifications to the current scripts.

This makes the algorithms universally available to other web caches, not only varnish.

Sysadmin Workflow

For a sysadmin to implement this and use it in a system he would need the following.

Prerequisites One varnish server and one web server.
Network connection between the varnish and web server.
Internet acesses to the varnish server.

Varnish A running varnish instance with the webserver specified as a backend in the standard varnish config.

Also one would need the small telnetscript.sh to implement the urls into a web cache rule.

Web server On the web server he would need access to the apache2 log file and the ability to run one of the proposed algorithm scripts. FBRTAA preferably.

Varnish worked great without a .vcl file containing any caching rules. (Negative) Negates original planning. Almost negated prototype.

Lack of understanding, research and clear documentation on how varnish works in correlation with the vcl language and default.vcl.
Lack of preparation and understanding of how varnish really worked.
There was insufficent time to test these each algorithms on other web caches.
This could be avoided by thourougher resourced and understood how something worked before starting with the approach. If documentation was not available different ways to achive knowledge around different aspects should be used like calling varnish, arranging a meeting with varnish, email, varnish irc channel or talking to somebody who already used the technology.

Varnish has proven to be a great product that is used by the both small and large company's in the business and IT world. Names like Verdens Gang, BBC, Vimeo and Wimp to name a few [15]. Varnish offers two products, a free basic version of varnish that offers the core functionality of varnish which is sufficient for many websites today.

Bigger company's usually takes advantage of the Varnish plus subscription which include services not included in the free version. Varnish plus

includes many great premade services and solutions like WebGui administration, statistics, mobile device detection, HTTP streaming, training discount and support from the varnish team.

This is in a way a Service Level Agreement from varnish that many company's prefer. But it comes with a great price, the 3 different subscriptions bronze, silver and gold have a starting price of 12000 eur/17000 usd/ 100 000 nok. For big companys this may be a small price to pay considering the potentional cost savings.

For smaller companys and web sites these prices are not an option and its from this perspective this thesis is based on. Helping sysadmins with implementing varnish in way which would reduce implementations time and management, making varnish more adoptable for smaller company's.

In a combination with varnish training where cost start from 1500 eur and this thesis, adoption of varnish for smaller companys will help getting the great benefits of the product varnish is.

Chapter 7

Conclusion

Automated web-cache configuration using machine learning

Approaching the problem statement in this thesis was something that needed alot of ideas and mindsets. What and how could one achieve automation regarding the configuration of web caches using machine learning.

Different approaches where suggested and used to solve this. One needed to find out the best way to do each aspect and then combine them somewhow to solve the problem.

By first learning and researching what was possible to do in each aspect one could start to see a way of combining machine learning and automation regarding the configuration of web caches. The way forward was to start simple and see solutions where available. Then trying to do each aspect alone by first seeing how a web cache was configured and see how it could be automated. After that the need to automaticly decide what one wanted to cache. And here machine learning was needed.

Finding the right way to figure out what kind of traffic one should cache was a challange to do. But after finding the right way of controlling what one wanted to cache and what not to cache helped see that a combination could be done by automating the configuration of web caches.

Experiments and results showed it possible to automaticly configure a web cache using a machine learning tecqhniue based on ranking traffic URLS, followed by configuring the web cache automaticly with caching rules contanting the ranked urls.

7.1 Future work

7.1.1 Combining TRRLA and FBRTAA

By combining the TRRLA and FBRTAA features into one would result in each URL getting an extra reward if it reoccurs often. The combination of having the TRRLA feature and FBRTAA feature would help give a more

accurate rank score over time.

7.1.2 Self-reconfiguration of cache policy

This feature could be added as a standalone script or as a sub routine that would incorporate all 3 proposed algorithms where it would change from running say the FBRLA to the TRRLA. If traffic pattern or other criteria that can be specified, this script could run the appropriate algorithm best suited.

7.1.3 Auto DDoS caching

Varnish itself has a built in DDOS function where it detects an anomaly in huge of amount of tcp syn floods and automatically rejects them. But not all other web caches have this function, web caches missing this kind of function could use these algorithms on a dedicated web cache to simply cache all the incoming requests to itself then rather sending it trough to the web server. Which would result in an automatic DDoS protection.

Bibliography

[1] Varnish. How does it work? https://www.varnish-software.com/what-is-varnish-plus, 2013.

[2] Adam Malone. Explaining varnish for beginners.http://www.acquia.com/blog/explaining-varnish-beginners, 2013.

[3] Admin. The incredible growth of web usage [1984-2013].The IncredibleGrowth of Web Usage [1984-2013], 2013.

[4] Mark Nottingham. Caching tutorial.http://www.mnot.net/cachedocs/, 2013.

[5] Per Buer. Withstanding ddos attacks with varnish and cots hard-ware. https://www.varnish-software.com/blog/withstanding-ddos-attacks-varnish-and-cots-hardware, 2013.

[6] Mike Wesser. Varnish cache as a ddos mitigation solution, withtips! http://www.mikewesson.com/2011/05/30/varnish-cache-as-a-ddos-mitigation-solution-with-tips/, 2011

[7] Mike Ciavarella Lee Damon. Habits of the highly effective system administra-tor. InHabits of the Highly Effective System Administrator, 2013

[8] Thomas A. Limoncelli.Time Managment for system administrators. OâReilly Me-dia, 2005

[9] Christina J. Hogan Thomas A. Limoncelli.The Practice of System and NetworkAdministration, Second Edition. Addison-Wesley Professional, 2007.

[10] Margaret Rouse. machine learning. http://whatis.techtarget.com/definition/machine-learning, 2011.

[11] Kristian Lyngstl, Per Buer, Dag-Erling Smrgrav, Poul-Henning Kamp.Varnish configuration language. https://www.varnish-cache.org/docs/3.0/reference/vcl.html, 2010

[12] Justin Zobel. Writing for computer science. http://www.amazon.com/Writing-Computer-Science-Justin-Zobel/dp/1852338024/ref=sr11?s=booksie=UTF8qid=1398272886sr=11, 2014.

[13] Nate Haug. Configuring varnish for high-availability with multiple webservers. http://www.lullabot.com/blog/article/configuring-varnish-high-availability-multiple-web-servers, 2011

[14] Joshua P. Mervine. performance-testing-with-httperf. http://mervine.net/performance-testing-with-httperf, 2011.

[15] Varnish inc. https://www.varnish-software.com/who-relies-on-varnish.https://www.varnish-software.com/who-relies-on-varnish, 2014

Chapter 8

Appendices

FBRLA.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Date::Parse;
use Storable;
use File::Copy;

# Hash variables
my %freq;
my $freqglobal;
my %P;

### Hit counts webserver ###

# Current time
my $U_time=time;
print "1.U_time: ";
print $U_time;
print "\n";


# Open time and save time stamp to file
open(TIME,"TIME.txt");
my $last_U_time = <TIME>;
chomp $last_U_time;
close TIME;

print "Last_U_time: " . $last_U_time . "\n";

# Read LOG timestamps and convert to EPOCHE
```

```
open(LOG, "/var/log/apache2/access.log");
print "Opening log file\n";

while ( my $line2 = <LOG> )
     {
#print "Reading log file\n";
if ($line2 =~ s/.*\[(.*)\].*/\1/g )
                # if ($line2 =~ s/.*\[(.*)\].*/\1/g )
#  if ( $line2 =~ /\[([^\]]*)\]/ )
                {
#print "REGEX matched: " . $line2;

 # convert timestamp to EPOCHE
                    my $O_time = str2time($line2);
 # print "O_time: " . $O_time . "\n";

   if ( $last_U_time < $O_time )

{
print "Found cutoff point for newer lines\n";
    while (my $line=<LOG>)
     {
        if ($line =~ /]\s".*\s(\/\S.+)\sHTTP/ )
                   {
                  my $url = $1;
# print "Found new URL: $url\n";

                $freq{$url}++;
                 $freqglobal++;
           }
        }
}




  }
}
open (TIMEFILE, ">TIME.txt");
print TIMEFILE $U_time . "\n";
close TIMEFILE;


print "U is populated, running algorithm\n";

FBRLA();
```

```
### FBRLA ###

sub FBRLA {

my $N = 20;  # Max URLs in %URLhash
my $AT = 2;  # Attrition value
my $ET = 4;  # Eviction Threshold

# open P file
print "################## Opening P.txt\n";


open (FILE,"P.txt");
while ( my $line = <FILE> ){
chomp $line;
( my $url, my $mana ) = split /:/,$line;
print "inserting into P: $url -> $mana\n";
        $P{$url} = $mana;
}
close(FILE);



# Update Importance #
print "################## Updating importance\n";

foreach my $key ( keys %P ){
print "reducing $key rank from $P{$key} to " .
( $P{$key} - $AT) . "\n";
$P{$key} = $P{$key} - $AT;
}

### Eviction Treshold ###

    print "################## Eviction \n";
my $evicted = 0;
    if ( keys %P >= $N ){
foreach my $key ( keys %P ){
    if ( $P{$key} < $ET ){
print "URL $key has importance below $ET ( $P{$key} ),
evicting\n";
delete $P{$key};
$evicted++;
    }
}
```

```
    } else {
print "Not evicting, P not large enough: " . (scalar keys %P)
. "\n";
    }



### Add new URLS ###

my $added = 0;
foreach my $url (sort { $freq{$b} <=> $freq{$a} } keys %freq)
{
    # print "$url -> $freq{$url}\n";
    # print FILE "$url -> $freq{$url}\n";
    if ( scalar keys %P < $N ){
$P{$url} = int($freq{$url} / $freqglobal * 1000);
$added++;
print "adding new URL $url with rank $P{$url}, ( a: $added,
e: $evicted) used slots: " . (scalar keys %P) ." \n";
    }
}


### Save P to file ###

open(P,">P.txt");
foreach my $key ( keys %P ){
print P "$key:$P{$key}\n";
}
close(P);
}

### end of sub FBRLA ###
print "Algorithm end\n";




### Make .vcl file ###

my $midstart = 'if (req.request == "GET" && (';
my $midstring;
foreach my $key (keys %P){
    $midstring .= "req.url == \"$key\" || ";
}
```

```
$midstring =~ s/\|\| $//;
my $midend = ')) {';
$midstring = $midstart . $midstring . $midend;

# print "midstring: $midstring\n";

open(MID,">mid.vcl");
print MID $midstring . "\n";
close(MID);

system("cat head.vcl mid.vcl tail.vcl > new.vcl");
print "Generated new.vcl\n";


### SCP new.vcl to varnish server ###

system("scp new.vcl ubuntu\@10.0.4.2: ");
print "Transferred file\n";



### Varnish command via ssh connection


# Use default vcl
system("ssh root\@10.0.4.2 /home/ubuntu/FBRLA_varnish_telnet
.sh" );
print "Running varnish telnet commands";


## Copy P.txt incrementaly to EXPERIMENT_2 folder ##

my $name = "P.txt";
if (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/FBRLA/P_files/
$name") {
    my $num = 1;
    $num ++ while (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/
    FBRLA/P_files/$name\_$num");
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/FBRLA/P_files/$name\_$num")
    or die "cannot copy file";
} else {
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/FBRLA/P_files/$name") or
    die "cannot copy file 2";
}
```

```
exit 0;
```

TRRLA.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Date::Parse;
use Storable;
use File::Copy;

# Hash variables
my %freq;
my $freqglobal;
my %P;
my @EvictedARRAY;

# Current time
my $U_time=time;
print "1.U_time: ";
print $U_time;
print "\n";


# Evicted time
my $E_time=time;


# Open time and save time stamp to file
open(TIME,"TIME.txt");
my $last_U_time = <TIME>;
chomp $last_U_time;
close TIME;

print "Last_U_time: " . $last_U_time . "\n";

# Read LOG timestamps and convert to EPOCHE
open(LOG, "/var/log/apache2/access.log");
print "Opening log file\n";

while ( my $line2 = <LOG> )
     {
#print "Reading log file\n";
if ($line2 =~ s/.*\[(.*)\].*/\1/g )
                # if ($line2 =~ s/.*\[(.*)\].*/\1/g )
#  if ( $line2 =~ /\[([^\]]*)\]/ )
                {
```

```
#print "REGEX matched: " . $line2;

 # convert timestamp to EPOCHE
                    my $O_time = str2time($line2);
 # print "O_time: " . $O_time . "\n";

   if ( $last_U_time < $O_time )

{
print "Found cutoff point for newer lines\n";
    while (my $line=<LOG>)
     {
        if ($line =~ /]\s".*\s(\/\S.+)\sHTTP/ )
                   {
                  my $url = $1;
# print "Found new URL: $url\n";

                $freq{$url}++;
                 $freqglobal++;
           }
        }
}




  }
}
open (TIMEFILE, ">TIME.txt");
print TIMEFILE $U_time . "\n";
close TIMEFILE;


print "U is populated, running algorithm\n";

TRRLA();


### TRRLA ###

sub TRRLA {

my $N = 20;  # Max URLs in %URLhash
my $AT = 2;  # Attrition value
my $ET = 4;  # Eviction Threshold
```

```
# open P file
print "################## Opening P.txt\n";

open (FILE,"P.txt");
while ( my $line = <FILE> ){
chomp $line;
( my $url, my $mana ) = split /:/,$line;
print "inserting into P: $url -> $mana\n";
        $P{$url} = $mana;
}
close (FILE);


## Standard mana value ##
my $newmana = 10;
# my $newmana = int($freq{$url} / $freqglobal * 100);


# Update Importance #
print "################## Updating importance\n";

foreach my $key ( keys %P ){
print "reducing $key rank from $P{$key} to " . ( $P{$key} -
$AT) . "\n";
$P{$key} = $P{$key} - $AT;
}



### Eviction Treshold ###

print "################## Eviction \n";

open (EFILETIME, ">Evicted_time_urls.txt");

my $evicted = 0;
    if ( keys %P >= $N ){
foreach my $key ( keys %P ){
    if ( $P{$key} < $ET ){
print "URL $key has importance below $ET ( $P{$key} ),
evicting\n";

# Print time and url to file
print EFILETIME $E_time . "," . $key . "\n";

delete $P{$key};
```

```
$evicted++;
    }
}
    } else {
print "Not evicting, P not large enough: " . (scalar keys %P)
. "\n";
    }
close (EFILETIME);



### Add new URLS ###
my $added = 0;
foreach my $url ( sort { $freq{$b} <=> $freq{$a} } keys %freq)
{
    # grep url from evicted file
    if ( scalar keys %P < $N ){
my $EvictedURLs = `cat Evicted_time_urls.txt | grep $url |
tail -1`;

# Get last evicted url TIMESTAMP
if($EvictedURLs){
    @EvictedARRAY = split (",", $EvictedURLs);
    my $interval = (($E_time - $EvictedARRAY[0]) / 20);

   # new mana
    my $newmana = int($freq{$url} / $freqglobal * 1000);
    $newmana = int($freq{$url} / $freqglobal * 1000) *
    (1/$interval) if $interval > 0;

    $P{$url} = $newmana;
} else {
    $P{$url} = int($freq{$url} / $freqglobal * 1000);
    print "No eviction time, using standard mana 10";
}
$added++;
print "adding new URL $url with rank $P{$url}, ( a: $added,
e: $evicted) used slots: " . (scalar keys %P) ." \n";
    }



}
```

```
### Save P to file ###

open(P,">P.txt");
foreach my $key ( keys %P ){
print P "$key:$P{$key}\n";
}
close(P);
}

### end of sub TRRLA ###
print "Algorithm end\n";




### Make .vcl file ###

my $midstart = 'if (req.request == "GET" && (';
my $midstring;
foreach my $key (keys %P){
    $midstring .= "req.url == \"$key\" || ";
}
$midstring =~ s/\|\| $//;
my $midend = ')) {';
$midstring = $midstart . $midstring . $midend;

# print "midstring: $midstring\n";

open(MID,">mid.vcl");
print MID $midstring . "\n";
close(MID);

system("cat head.vcl mid.vcl tail.vcl > TRRLA_new.vcl");
print "Generated TRRLA_new.vcl\n";


### SCP TRRLA_new.vcl to varnish server ###
system("scp TRRLA_new.vcl ubuntu\@10.0.4.2: ");
print "Transferred file\n";



# Run telnetscript on varnish
system("ssh root\@10.0.4.2 /home/ubuntu/
TRRLA_varnish_telnet.sh " );
```

```
print "Running varnish telnet commands";



# print freqglobal
# print "Global: " . $freqglobal . "\n";

## Copy P.txt incrementaly to EXPERIMENT_2 folder ##

my $name = "P.txt";
if (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/TRRLA/P_files/
$name") {
    my $num = 1;
    $num ++ while (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/
    TRRLA/P_files/$name\_$num");
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/TRRLA/P_files/$name\_$num") or
    die "cannot copy file";
} else {
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/TRRLA/P_files/$name") or die
    "cannot copy file 2";
}

exit 0;
```

FBRTAA.pl

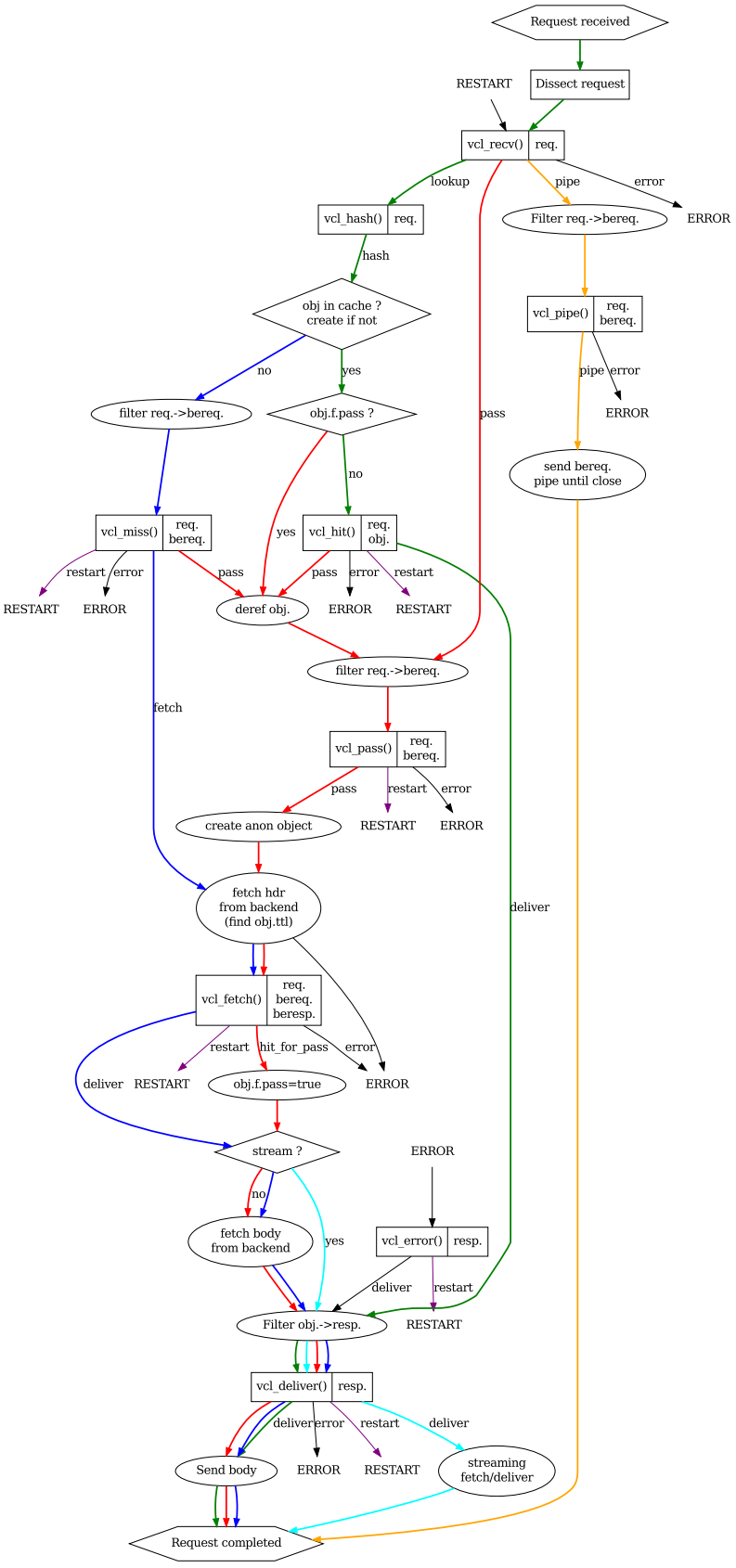
```
#!/usr/bin/perl

use strict;
use warnings;
use Date::Parse;
use Storable;

# Hash variables
my %freq;
my $freqglobal;
my %P;

### Hit counts webserver ###

# Current time
my $U_time=time;
print "1.U_time: ";
print $U_time;
print "\n";


# Open time and save time stamp to file
open(TIME,"TIME.txt");
my $last_U_time = <TIME>;
chomp $last_U_time;
close TIME;

print "Last_U_time: " . $last_U_time . "\n";

# Read LOG timestamps and convert to EPOCHE
open(LOG, "/var/log/apache2/access.log");
print "Opening log file\n";

while ( my $line2 = <LOG> )
     {
#print "Reading log file\n";
if ($line2 =~ s/.*\[(.*)\].*/\1/g )
                # if ($line2 =~ s/.*\[(.*)\].*/\1/g )
#  if ( $line2 =~ /\[([^\]]*)\]/ )
                {
#print "REGEX matched: " . $line2;

 # convert timestamp to EPOCHE
                    my $O_time = str2time($line2);
```

```
 # print "O_time: " . $O_time . "\n";

   if ( $last_U_time < $O_time )

{
print "Found cutoff point for newer lines\n";
    while (my $line=<LOG>)
     {
        if ($line =~ /]\s".*\s(\/\S.+)\sHTTP/ )
                   {
                  my $url = $1;
# print "Found new URL: $url\n";

                 $freq{$url}++;
                  $freqglobal++;
           }
        }
}




  }
}
open (TIMEFILE, ">TIME.txt");
print TIMEFILE $U_time . "\n";
close TIMEFILE;


print "U is populated, running algorithm\n";

FBRTAA();


### FBRTAA ###

sub FBRLA {

my $N = 20;  # Max URLs in %URLhash
my $AT = 2;  # Attrition value
my $ET = 4;  # Eviction Threshold

# open P file
print "################## Opening P.txt\n";
```

```
open (FILE,"P.txt");
while ( my $line = <FILE> ){
chomp $line;
( my $url, my $mana ) = split /:/,$line;
print "inserting into P: $url -> $mana\n";
        $P{$url} = $mana;
}
close(FILE);

# Update Importance #
print "################## Updating importance\n";

foreach my $key ( keys %P ){
print "reducing $key rank from $P{$key} to " . ( $P{$key}
- 100) . "\n";
$P{$key} = $P{$key} - $AT;
}

### Eviction Treshold ###

    print "################## Eviction \n";
my $evicted = 0;
    if ( keys %P >= $N ){
foreach my $key ( keys %P ){
    if ( $P{$key} < $ET ){
print "URL $key has importance below $ET ( $P{$key} ),
evicting\n";
delete $P{$key};
$evicted++;
    }
}
    } else {
print "Not evicting, P not large enough: " . (scalar keys %P)
. "\n";
    }



### Add new URLS ###

my $added = 0;
foreach my $url ( sort { $freq{$b} <=> $freq{$a} } keys %freq)
{
    # print "$url -> $freq{$url}\n";
    # print FILE "$url -> $freq{$url}\n";
    if ( scalar keys %P < $N ){
```

```
my $interval = (($E_time - $EvictedARRAY[0]) / 20);
$P{$url} = int($freq{$url} / $freqglobal * 1000) *
(1/($U_time - $interval) if $interval > 0;
$added++;
print "adding new URL $url with rank $P{$url}, ( a: $added,
e: $evicted) used slots: " . (scalar keys %P) ." \n";
    }
}


### Save P to file ###

open(P,">P.txt");
foreach my $key ( keys %P ){
print P "$key:$P{$key}\n";
}
close(P);
}

### end of sub FBRLA ###
print "Algorithm end\n";




### Make .vcl file ###

my $midstart = 'if (req.request == "GET" && (';
my $midstring;
foreach my $key (keys %P){
    $midstring .= "req.url == \"$key\" || ";
}
$midstring =~ s/\|\| $//;
my $midend = ')) {';
$midstring = $midstart . $midstring . $midend;

# print "midstring: $midstring\n";

open(MID,">mid.vcl");
print MID $midstring . "\n";
close(MID);

system("cat head.vcl mid.vcl tail.vcl > new.vcl");
print "Generated new.vcl\n";
```

```
### SCP new.vcl to varnish server ###

system("scp new.vcl ubuntu\@10.0.4.2: ");
print "Transferred file\n";



### Varnish command via ssh connection


# Use default vcl
system("ssh root\@10.0.4.2 /home/ubuntu/telnetvarnish.sh " );
print "Running varnish telnet commands";


# print freqglobal
# print "Global: " . $freqglobal . "\n";


## Copy P.txt incrementaly to EXPERIMENT_2 folder ##

my $name = "P.txt";
if (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/FBRTAA/P_files/
$name") {
    my $num = 1;
    $num ++ while (-e "/home/ubuntu/OK-Scripts/EXPERIMENTS_2/
    FBRTAA/P_files/$name\_$num");
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/FBRTAA/P_files/$name\_$num")
    or die "cannot copy file";
} else {
    copy("/home/ubuntu/OK-Scripts/$name","/home/ubuntu/
    OK-Scripts/EXPERIMENTS_2/FBRTAA/P_files/$name") or
    die "cannot copy file 2";
}


exit 0;
```


Request received
RESTART
Dissect request
vcl_recv()
req.
lookup
pipe
error
ERROR
vcl_hash()
req.
Filter req.->bereq.
hash
obj in cache ?
create if not
vcl_pipe()
req.
bereq.
no
yes
pipe
error
ERROR
filter req.->bereq.
obj.f.pass ?
pass
send bereq.
pipe until close
no
vcl_miss()
req.
bereq.
yes
vcl_hit()
req.
obj.
restart
error
pass
pass
error
restart
RESTART
ERROR
deref obj.
ERROR
RESTART
filter req.->bereq.
fetch
vcl_pass()
req.
bereq.
pass
restart
error
create anon object
RESTART
ERROR
fetch hdr
from backend
(find obj.ttl)
deliver
vcl_fetch()
req.
bereq.
beresp.
restart
hit_for_pass
error
deliver
RESTART
obj.f.pass=true
ERROR
stream ?
ERROR
no
fetch body
from backend
yes
vcl_error()
resp.
deliver
restart
Filter obj.->resp.
RESTART
vcl_deliver()
resp.
deliver
error
restart
deliver
Send body
ERROR
RESTART
streaming
fetch/deliver
Request completed

Varnish logic